

Faster Algorithms for Online Topological Ordering

Telikepalli Kavitha Rogers Mathew
 Indian Institute of Science
 Bangalore, India
 {kavitha, rogers}@csa.iisc.ernet.in

Abstract

We present two algorithms for maintaining the topological order of a directed acyclic graph with n vertices, under an online edge insertion sequence of m edges. Efficient algorithms for online topological ordering have many applications, including online cycle detection, which is to discover the first edge that introduces a cycle under an arbitrary sequence of edge insertions in a directed graph. The current fastest algorithms for the online topological ordering problem run in time $O(\min(m^{3/2} \log n, m^{3/2} + n^2 \log n))$ and $O(n^{2.75})$ (the latter algorithm is faster for dense graphs, i.e., when $m > n^{11/6}$). In this paper we present faster algorithms for this problem.

We first present a simple algorithm with running time $O(n^{5/2})$ for the online topological ordering problem. This is the current fastest algorithm for this problem on dense graphs, i.e., when $m > n^{5/3}$. We then present an algorithm with running time $O((m + n \log n) \sqrt{m})$, which is an improvement over the $O(\min(m^{3/2} \log n, m^{3/2} + n^2 \log n))$ algorithm - it is a *strict* improvement when m is sandwiched between $\omega(n)$ and $O(n^{4/3})$. Our results yield an improved upper bound of $O(\min(n^{5/2}, (m + n \log n) \sqrt{m}))$ for the online topological ordering problem.

1 Introduction

Let $G = (V, E)$ be a directed acyclic graph (DAG) with $|V| = n$ and $|E| = m$. In a topological ordering, each vertex $v \in V$ is associated with a value $\text{ord}(v)$ such that for each directed edge $(u, v) \in E$ we have $\text{ord}(u) < \text{ord}(v)$. When the graph G is known in advance (i.e., in an offline setting), there exist well-known algorithms to compute a topological ordering of G in $O(m + n)$ time [4].

In the *online* topological ordering problem, the edges of the graph G are not known in advance but are given one at a time. We are asked to maintain a topological ordering of G under these edge insertions. That is, each time an edge is added to G , we are required to update the function ord so that for all the edges (u, v) in G , it holds that $\text{ord}(u) < \text{ord}(v)$. The naïve way of maintaining an online topological order, which is to compute the order each time from scratch with the offline algorithm, takes $O(m^2 + mn)$ time. However such an algorithm is too slow when the number of edges, m , is large. Faster algorithms are known for this problem (see Section 1.1). We show the following results here¹.

Theorem 1 *An online topological ordering of a directed acyclic graph G on n vertices, under a sequence of arbitrary edge insertions, can be computed in time $O(n^{5/2})$, independent of the number of edges inserted.*

Theorem 2 *An online topological ordering of a directed acyclic graph G on n vertices, under any sequence of insertions of m edges, can be computed in time $O((m + n \log n) \sqrt{m})$.*

The online topological ordering problem has several applications and efficient algorithms for this problem are used in online cycle detection routine in pointer analysis [11] and in incremental evaluation of computational circuits [2]. This problem has also been studied in the context of compilation [8, 9] where dependencies between modules are maintained to reduce the amount of recompilation performed when an update occurs.

¹Here and in the rest of the paper, we make the usual assumption that m is $\Omega(n)$ and there are no parallel edges, so m is $O(n^2)$.

1.1 Previous Results

The online topological ordering problem has been well-studied. Marchetti-Spaccamela et al. [7] gave an algorithm that can insert m edges in $O(mn)$ time. Alpern et al. [2] proposed a different algorithm which runs in $O(\|\delta\| \log \|\delta\|)$ time per edge insertion with $\|\delta\|$ measuring the number of edges of the minimal vertex subgraph that needs to be updated. However, note that not all the edges of this subgraph need to be updated and hence even $\|\delta\|$ time per insertion is not optimal. Katriel and Bodlaender [6] analyzed a variant of the algorithm in [2] and obtained an upper bound of $O(\min\{m^{3/2} \log n, m^{3/2} + n^2 \log n\})$ for a general DAG. In addition, they show that their algorithm runs in time $O(mk \log^2 n)$ for a DAG for which the underlying undirected graph has a treewidth of k , and they show an optimal running time of $O(n \log n)$ for trees. Pearce and Kelly [10] present an algorithm that empirically performs very well on sparse random DAGs, although its worst case running time is inferior to [6].

Ajwani et al. [1] gave the first $o(n^3)$ algorithm for the online topological ordering problem. They propose a simple algorithm that works in time $O(n^{2.75} \sqrt{\log n})$ and $O(n^2)$ space, thereby improving upon the algorithm [6] for dense DAGs. With some simple modifications in their data structure, they get $O(n^{2.75})$ time and $O(n^{2.25})$ space. They also demonstrate empirically that their algorithm outperforms the algorithms in [10, 7, 2] on a certain class of hard sequence of edge insertions.

The only non-trivial lower bound for online topological ordering is due to Ramalingam and Reps [12], who showed that an adversary can force any algorithm to perform $\Omega(n \log n)$ vertex relabeling operations while inserting $n - 1$ edges (creating a chain). There is a large gap between the lower bound of $\Omega(m + n \log n)$ and the upper bound of $O(\min\{n^{2.75}, m^{3/2} \log n, m^{3/2} + n^2 \log n\})$.

Our Results. The contributions of our paper are as follows:

- Theorem 1 shows an upper bound of $O(n^{5/2})$ for the online topological ordering problem. This is always better than the previous best upper bound of $O(n^{2.75})$ in [1] for dense graphs. Our $O(n^{5/2})$ algorithm is the current fastest algorithm for online topological ordering when $m > n^{5/3}$.
- Theorem 2 shows another improved upper bound of $O((m + n \log n) \sqrt{m})$. This improves upon the bounds of $O(m^{3/2} \log n)$ and $O(m^{3/2} + n^2 \log n)$ given in [6]. Note that this is a strict improvement over $\Theta(\min(m^{3/2} \log n, (m^{3/2} + n^2 \log n)))$ when m is sandwiched between $\omega(n)$ and $O(n^{4/3})$.

Combining our two algorithms, we have an improved upper bound of $O(\min(n^{5/2}, (m + n \log n) \sqrt{m}))$ for the online topological ordering problem.

Our $O(n^{5/2})$ algorithm is very simple and basically involves traversing successive locations of an array and checking the entries of the adjacency matrix; it uses no special data structures and is easy to implement, so it would be an efficient online cycle detection subroutine in practice also. The tricky part here is showing the bound on its running time and we use a result from [1] in its analysis. Our $O((m + n \log n) \sqrt{m})$ algorithm is an adaptation of the Katriel-Bodlaender algorithm in [6] (in turn based on the algorithm in [2]) and uses the Dietz-Sleator ordered list data structure and Fibonacci heaps.

Organization of the paper. In Section 2 we present our $O(n^{5/2})$ algorithm and show its correctness. We analyze its running time in Section 3. We present our $O((m + n \log n) \sqrt{m})$ algorithm and its analysis in Section 4. The missing details are given in the Appendix.

2 The $O(n^{5/2})$ Algorithm

We have a directed acyclic graph G on vertex set V . In this section we present an algorithm that maintains a bijection $\text{ord} : V \rightarrow \{1, 2, \dots, n\}$ which is our topological ordering. Let us assume that the graph is initially the empty graph and so any bijection from V to $\{1, 2, \dots, n\}$ is a valid topological ordering of V at the onset.

New edges get inserted to this graph and after each edge insertion, we want to update the current bijection from V to $\{1, 2, \dots, n\}$ to a valid topological ordering.

Let the function ord from V onto $\{1, 2, \dots, n\}$ denote our topological ordering. We also have the inverse function of ord stored as an array $A[1..n]$, where $A[i]$ is the vertex x such that $\text{ord}(x) = i$. We have the $n \times n$ 0-1 adjacency matrix M of the directed acyclic graph, corresponding to the edges inserted so far; $M[x, y] = 1$ if and only if there is an edge directed from vertex x to vertex y .

When a new edge (u, v) is added to G , there are two cases: (i) either $\text{ord}(u) < \text{ord}(v)$ in which case the current ordering ord is still a valid ordering, so we need to do nothing, except set the entry $M[u, v]$ to 1, or (ii) $\text{ord}(u) > \text{ord}(v)$ in which case we need to update ord . We now present our algorithm to update ord , when an edge (u, v) such that $\text{ord}(u) > \text{ord}(v)$, is added to G . If (u, v) creates a cycle, the algorithm quits; else it updates ord to a valid topological ordering.

[Let $s \rightsquigarrow t$ indicate that there is a directed path (perhaps, of length 0) from vertex s to vertex t in G . If $s \rightsquigarrow t$, we say s is an *ancestor* of t and t is a *descendant* of s . We use $s \rightarrow t$ to indicate that $(s, t) \in E$.]

2.1 Our algorithm for inserting a new edge (u, v) where $\text{ord}(u) > \text{ord}(v)$

Let $\text{ord}(v) = i$ and $\text{ord}(u) = j$. Our algorithm works only on the subarray $A[i..j]$ and computes a subset (call it ANC) of ancestors of u in $A[i..j]$ and a subset (call it DES) of descendants of v in $A[i..j]$; it assigns new positions in A to the vertices in $\text{ANC} \cup \text{DES}$. This yields the new topological ordering ord .

We describe our algorithm in two phases: Phase 1 and Phase 2. In Phase 1 we construct the sets:

$$\begin{aligned} \text{DES} &= \{y : i \leq \text{ord}(y) \leq t \text{ and } v \rightsquigarrow y\} \text{ (the set of descendants of } v \text{ in the subarray } A[i..t]) \\ \text{ANC} &= \{w : t \leq \text{ord}(w) \leq j \text{ and } w \rightsquigarrow u\} \text{ (the set of ancestors of } u \text{ in the subarray } A[t..j]) \end{aligned}$$

where t is a number such that $i \leq t \leq j$ and t has the following property: *if G is a DAG, then the number of descendants of v in $A[i..t]$ is exactly equal to the number of ancestors of u in $A[(t+1)..j]$.*

We then check if $(x, y) \in E$ for any $x \in \text{DES}$ and $y \in \text{ANC}$, or if $A[t] \in \text{DES} \cap \text{ANC}$. If either of these is true, then the edge (u, v) creates a cycle and G is no longer a DAG, so our algorithm quits. Else, we delete the elements of $\text{DES} \cup \text{ANC}$ from their locations in A , thus creating empty locations in A .

Phase 2 deals with inserting elements of ANC in $A[i..t]$ and the elements of DES in $A[(t+1)..j]$. Note that we cannot place the vertices in ANC straightaway in the empty locations previously occupied by DES in $A[i..t]$ since there might be further ancestors of elements of ANC in $A[i..t]$. Similarly, there might be descendants of elements of DES in $A[(t+1)..j]$. Hence we need Phase 2 to add more elements to ANC and to DES , and to insert elements of ANC in $A[i..t]$ and those of DES in $A[(t+1)..j]$ correctly.

2.1.1 PHASE 1.

We now describe Phase 1 of our algorithm in detail. Initially the set $\text{ANC} = \{u\}$ and the set $\text{DES} = \{v\}$. We maintain ANC and DES as queues. We move a pointer LeftPtr from location j leftwards (towards i

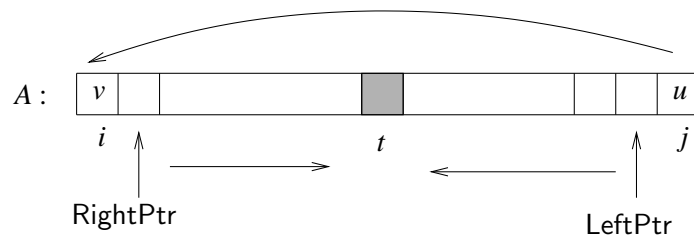


Figure 1: The pointer LeftPtr and RightPtr meet at the location t .

as shown in Figure 1) in order to find an ancestor w of u , which gets added to the end of the queue ANC . Then we move a pointer RightPtr from location i rightwards to find a descendant y of v , which gets added

to the end of DES. Then we go back to LeftPtr, and thus interleave adding a vertex to ANC with adding a vertex to DES so that we balance the size of ANC constructed so far with the size of DES. When LeftPtr and RightPtr meet, that defines our desired location t . We present the detailed algorithm for Phase 1 as Algorithm 2.1. [If $(x, y) \in E$, we say x is a *predecessor* of y and y is a *successor* of x .]

Algorithm 2.1 Our algorithm to construct the sets ANC and DES in Phase 1.

```

– Initialize  $ANC = \{u\}$  and  $DES = \{v\}$ .
– set  $RightPtr = i$  and  $LeftPtr = j$ . {So  $LeftPtr > RightPtr$ .}
while TRUE do
     $LeftPtr = LeftPtr - 1$ ;
    while  $LeftPtr > RightPtr$  and  $A[LeftPtr]$  is not a predecessor of any vertex in ANC do
         $LeftPtr = LeftPtr - 1$ ;
    end while
    if  $A[LeftPtr]$  is a predecessor of some vertex in ANC then
        – Insert the vertex  $A[LeftPtr]$  to the queue ANC.
    end if
    if  $LeftPtr = RightPtr$  then
        break {this makes the algorithm break the while TRUE loop}
    end if
    {Now the symmetric process from the side of  $v$ .}
     $RightPtr = RightPtr + 1$ ;
    while  $RightPtr < LeftPtr$  and  $A[RightPtr]$  is not a successor of any vertex in DES do
         $RightPtr = RightPtr + 1$ ;
    end while
    if  $A[RightPtr]$  is a successor of some vertex in DES then
        – Insert the vertex  $A[RightPtr]$  to the queue DES.
    end if
    if  $RightPtr = LeftPtr$  then
        break
    end if {If the algorithm does not break the while TRUE loop, then  $LeftPtr > RightPtr$ .}
end while

```

The above algorithm terminates when $LeftPtr = RightPtr$ is satisfied. Set t to be this location: that is, $t = RightPtr = LeftPtr$. It is easy to check that Algorithm 2.1 constructs the sets:

$$DES = \{y : i \leq \text{ord}(y) \leq t \text{ and } v \rightsquigarrow y\}; \quad ANC = \{w : t \leq \text{ord}(w) \leq j \text{ and } w \rightsquigarrow u\}.$$

That is, DES is the set of descendants of v in $A[i..t]$ and ANC is the set of ancestors of u in $A[t..j]$. The following lemma is straightforward.

Lemma 1 *If the new edge (u, v) creates a cycle, then (i) either $A[t] \in DES \cap ANC$ or (ii) there is some $x \in DES$ and $y \in ANC$ such that there is an edge from x to y .*

Proof: If a cycle is created by the insertion of the edge (u, v) , then it implies that $v \rightsquigarrow u$ in the current graph. That is, there is a directed path p in the graph from v to u , before (u, v) was inserted. This implies that there is either an element in $DES \cap ANC$ or there is an edge in p that connects a descendant of v in $A[i..t]$ to an ancestor of u in $A[t..j]$ (see Figure 2).

Hence either there is an element in $DES \cap ANC$, the only such element can be $A[t]$, or there has to be an edge from a vertex in DES to a vertex in ANC, where DES is the set of descendants of v in $A[i..t]$ and ANC is the set of ancestors of u in $A[t..j]$. ■

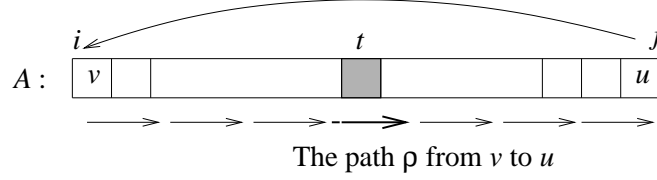


Figure 2: There has to be an edge in ρ from a descendant of v in $A[i..t]$ to an ancestor of u in $A[t..j]$ or $A[t] \in \text{ANC} \cap \text{DES}$.

Lemma 1 shows us that cycle detection is easy, so let us assume henceforth that the edge (u, v) does not create a cycle, hence the graph G is still a DAG. The following lemma shows that the number t has the desired property that we were looking for.

Lemma 2 *The number t has the property that the number of descendants of v in $A[i..t]$ is equal to the number of ancestors of u in $A[(t+1)..j]$.*

Proof: We have $t = \text{RightPtr} = \text{LeftPtr}$. When we terminate the while loop, ANC is the set of ancestors of u in $A[t..j]$ and DES is the set of descendants of v from $A[t..j]$. Since (u, v) does not create a cycle, $A[t]$ is either a descendant of v or an ancestor of u , but not both.

If the main while loop got broken because of the first break statement, then $A[t]$ is a descendant of v and if the main while loop got broken due to the second break statement, then $A[t]$ is an ancestor of u . Since we interleave adding a vertex to the set ANC with adding a vertex to the set DES , if we break because of the first break statement, we have $|\text{ANC}| = |\text{DES}|$; and if we break because of the second break statement, we have $|\text{ANC}| = |\text{DES}| + 1$. Thus in both cases, it holds that the number of descendants of v in $A[i..t]$ is equal to the number of ancestors of u in $A[(t+1)..j]$. ■

At the end of Phase 1, since G is still a DAG, we delete all the elements of $\text{ANC} \cup \text{DES}$ from their current locations in A . The vertices in ANC have to find new places in $A[i..t]$ and the vertices in DES have to find new places in $A[(t+1)..j]$.² This forms Phase 2 of our algorithm.

PHASE 2. We now describe Phase 2 from the side of ANC as Algorithm 2.2. (A symmetric procedure also takes place on the side of DES .) In this phase vertices get deleted from ANC and vertices can also get added to ANC .

Algorithm 2.2 Phase 2 of our algorithm from the side of ANC .

```

LeftPtr = t;
while LeftPtr ≥ i do
  if A[LeftPtr] is an empty location then
    – delete the head of the queue ANC, call it  $h$ , and set  $A[\text{LeftPtr}] = h$ 
  else if A[LeftPtr] (call it  $r$ ) is a predecessor of some element in ANC then
    – insert  $r$  to the queue ANC
    – delete the head of the queue ANC, call it  $h$ , and set  $A[\text{LeftPtr}] = h$ 
  end if
  LeftPtr = LeftPtr – 1;
end while

```

In Phase 2, the pointer LeftPtr traverses the subarray $A[t..i]$ (from t leftwards to i) and elements get deleted/inserted in ANC . Whenever LeftPtr sees an empty location in A , the head of the queue ANC is deleted from ANC and is assigned to that empty location. Whenever $A[\text{LeftPtr}]$ is a predecessor of some

²Note that the number of empty locations in $A[i..t]$ exactly equals $|\text{ANC}|$ and the number of empty locations in $A[(t+1)..j]$ exactly equals $|\text{DES}|$.

element in ANC, then $A[\text{LeftPtr}]$ is removed from that location and is inserted into ANC and the current head of the queue ANC is inserted into that location. We have the following lemma, which is simple to show. Its proof is included in the Appendix.

Lemma 3 *The subroutine in Algorithm 2.2 maintains the following invariant in every iteration of the while loop: the number of elements in ANC equals the number of empty locations in the subarray $A[i..\text{LeftPtr}]$.*

Proof: It is easy to see that the invariant is true at the beginning of the subroutine. In other words, at the end of Phase 1, the number of empty locations in $A[i..t]$ exactly equals $|\text{ANC}|$.

We will now show that this invariant is maintained throughout Phase 2. Whenever LeftPtr sees an empty location in A , we delete an element from ANC, hence this invariant is maintained since the number of empty locations in $A[i..\text{LeftPtr}]$ decreases by one and so does the size of ANC. Whenever LeftPtr sees a predecessor p of some element of ANC at the current location, then p is deleted from its current location $\text{LeftPtr} = \ell$ in A and p is inserted into ANC; the leading element h of ANC gets deleted from ANC and we assign $A[\ell] = h$. Thus neither the number of empty locations in $A[i..\text{LeftPtr}]$ nor the size of ANC changes by our deletion and insertion, so the invariant is maintained. Hence when we exit the while loop, which is when $\text{LeftPtr} = i - 1$, the queue ANC will be empty. ■

We then traverse the subarray $A[(t + 1)..j]$ from location $t + 1$ to location j and execute the algorithm in Algorithm 2.2 with respect to DES. For the sake of completeness, we present that algorithm as Algorithm 2.3 below.

Algorithm 2.3 Phase 2 of our algorithm from the side of DES.

```

RightPtr =  $t + 1$ ;
while RightPtr  $\leq j$  do
  if  $A[\text{RightPtr}]$  is an empty location then
    – delete the head of the queue DES, call it  $\ell$ , and set  $A[\text{RightPtr}] = \ell$ 
  else if  $A[\text{RightPtr}]$  (call it  $r$ ) is a successor of some element in DES then
    – insert  $r$  to the queue DES
    – delete the head of the queue DES, call it  $\ell$ , and set  $A[\text{RightPtr}] = \ell$ 
  end if
  RightPtr = RightPtr + 1;
end while

```

A lemma analogous to Lemma 3 will show that there is always enough room in the array $A[\text{RightPtr}..j]$ to accommodate the elements of DES (refer Algorithm 2.3). This completes the description of our algorithm to update the topological ordering when a new edge is inserted.

2.1.2 Correctness.

We would now like to claim that after running Phase 1 and Phase 2 of our algorithm, we have a valid topological ordering. Our ordering is defined in terms of the array A . Each element x that has been assigned a new location in A has a new ord value, which is the index of its new location. For elements that never belonged to $\text{ANC} \cup \text{DES}$, the ord value is unchanged. For the sake of clarity, let us call the ordering before the new edge (u, v) was inserted as ord and let ord' denote the new function after executing our algorithm. We will show the following theorem here (its proof is included in the Appendix).

Theorem 3 *The function ord' is a valid topological ordering.*

Proof: We need to show that ord' is a valid topological ordering. Consider any edge (x, y) in the graph. We will show that $\text{ord}'(x) < \text{ord}'(y)$. We will split this into three cases.

- $x \in \text{ANC}$. There are three further cases: (i) $y \in \text{ANC}$, (ii) $y \in \text{DES}$, (iii) y is neither in ANC nor in DES. In case (i), both x and y are in ANC and since there is an edge from x to y , the vertex y is ahead of x in the queue ANC. So y gets deleted from ANC before x and is hence assigned a higher indexed location in A than x . In other words, $\text{ord}'(x) < \text{ord}'(y)$. In case (ii), we have $\text{ord}'(x) \leq t < \text{ord}'(y)$. In case (iii), since elements of ANC move to lower indexed locations in A , we have $\text{ord}'(x) < \text{ord}(x)$ whereas $\text{ord}'(y) = \text{ord}(y)$; since $\text{ord}(x) < \text{ord}(y)$, we get $\text{ord}'(x) < \text{ord}'(y)$.
- $x \in \text{DES}$. There are only two cases here: (i) $y \in \text{DES}$ or (ii) y is neither in ANC nor in DES. This is because if $x \in \text{DES}$ and $y \in \text{ANC}$, then $y \rightsquigarrow u \rightarrow v \rightsquigarrow x$. So $(x, y) \in E$ creates a cycle. This is impossible since we assumed that after inserting the edge (u, v) , G remains a DAG. Thus we cannot have $x \in \text{DES}$ and $y \in \text{ANC}$ for $(x, y) \in E$.
In case (i) here, because $(x, y) \in E$, the vertex x is ahead of the vertex y in the queue DES, so x gets deleted from DES before y and is hence assigned a lower indexed location in A than y . Equivalently, $\text{ord}'(x) < \text{ord}'(y)$. In case (ii) here, since $x \in \text{DES}$ and $y \notin \text{DES}$, it has to be that either $\text{ord}(y) > \text{ord}(u)$ in which case $\text{ord}'(x) < \text{ord}(y) = \text{ord}'(y)$, or by the time RightPtr visits the location in A containing y , the vertex x is already deleted from the queue DES - otherwise, y would have been inserted into DES. Thus $\text{ord}'(x) < \text{ord}(y) = \text{ord}'(y)$.
- $x \notin \text{ANC} \cup \text{DES}$. There are three cases again here: (i) $y \in \text{ANC}$, (ii) $y \in \text{DES}$, (iii) y is neither in ANC nor in DES. The arguments here are similar to the earlier arguments and it is easy to check that in all three cases we have $\text{ord}'(x) < \text{ord}'(y)$.

Thus ord' is a valid topological ordering. ■

Thus our algorithm is correct. In Section 3 we will show that its running time, summed over all edge insertions, is $O(n^{5/2})$.

3 Running Time Analysis

The main tasks in our algorithm to update ord to ord' (refer to Algorithms 2.1, 2.2, and 2.3) are:

- (1) moving a pointer LeftPtr from location j to i in the array A and checking if $A[\text{LeftPtr}]$ is a *predecessor* of any element of ANC;
- (2) moving a pointer RightPtr from location i to j in the array A and checking if $A[\text{RightPtr}]$ is a *successor* of any element of DES;
- (3) checking at the end of Phase 1, if $(x, y) \in E$ for any x in $\{x : i \leq \text{ord}(x) \leq t \text{ and } v \rightsquigarrow x\}$ and any y in $\{y : t \leq \text{ord}(y) \leq j \text{ and } y \rightsquigarrow u\}$. (If so, then (u, v) creates a cycle.)

Lemma 4 bounds the cost taken by Step 3 over all iterations. It can be proved using a potential function argument.

Lemma 4 *The cost for task 3, stated above, summed over all edge insertions, is $O(n^2)$.*

Proof: We need to check if there is an edge (x, y) between some $x \in \text{DES}$ and some $y \in \text{ANC}$. We pay a cost of $|\text{ANC}| \cdot |\text{DES}|$ for checking $|\text{ANC}| \cdot |\text{DES}|$ many entries of the matrix M .

Case(i): There is no adjacent pair (x, y) for $x \in \text{DES}, y \in \text{ANC}$. Then the cost $|\text{DES}| \cdot |\text{ANC}|$ can be bounded by $N(e)$, which is the number of pairs of vertices (y, x) for which the relationship $y \rightsquigarrow x$ has started now for the first time (due to the insertion of the edge (u, v)). Recall that at the end of Phase 1, DES is the set $\{x : i \leq \text{ord}(x) \leq t \text{ and } v \rightsquigarrow x\}$ and ANC is the set $\{y : i \leq \text{ord}(y) \leq t \text{ and } y \rightsquigarrow u\}$. Thus each vertex in this set DES currently has a lower ord value than each vertex in ANC - so the only relationship that could have existed between such an x and y is $x \rightsquigarrow y$, which we have ensured does not exist. Thus these pairs (x, y)

were incomparable and now the relationship $y \rightsquigarrow x$ has been established. It is easy to see that $\sum_{e \in E} N(e)$ is at most $\binom{n}{2}$ since any pair of vertices can contribute at most 1 to $\sum_{e \in E} N(e)$.

Case(ii): There is indeed an adjacent pair (x, y) for $x \in \text{DES}, y \in \text{ANC}$. Then we quit, since G is no longer a DAG. The check that showed G to contain a cycle cost us $|\text{ANC}| \cdot |\text{DES}|$, which is $O(n^2)$. We pay this cost only once as this is the last step of the algorithm. ■

Let ord_e be our valid topological ordering before inserting edge e and let ord'_e be our valid topological ordering after inserting e . Lemma 5 is our first step in bounding the cost taken for tasks 1 and 2 stated above.

Lemma 5 *The cost taken for tasks 1 and 2, stated above, is $\sum_{x \in V} |\text{ord}_e(x) - \text{ord}'_e(x)|$.*

Proof: In Step 1 we find out if $A[\text{LeftPtr}] = x$ is a predecessor of any element of ANC by checking the entries $M[x, w]$ for each $w \in \text{ANC}$. Each element w which is currently in ANC pays unit cost for checking the entry $M[x, w]$.

Any element $w \in \text{ANC}$ belongs to the set ANC while LeftPtr moves from location $\text{ord}_e(w)$ to $\text{ord}'_e(w)$. When $\text{LeftPtr} = \text{ord}_e(w)$ and we identify $A[\text{ord}_e(w)] = w$ to be a predecessor of some element in ANC, the vertex w gets inserted into ANC. When LeftPtr is at some empty location β and the vertex w is the head of the queue ANC, then w is deleted from ANC and we set $A[\beta] = w$, which implies that $\text{ord}'_e(w) = \beta$. So the total cost paid by w is $\text{ord}_e(w) - \text{ord}'_e(w)$, which is to check the entries $M[A[\text{LeftPtr}], w]$ as the pointer LeftPtr moves from location $\text{ord}_e(w) - 1$ to $\text{ord}'_e(w)$.

Symmetrically, for any vertex y that belonged to DES during the course of the algorithm, the cost paid by y is $\text{ord}'_e(y) - \text{ord}_e(y)$. A vertex z that never belonged to $\text{ANC} \cup \text{DES}$, does not pay anything and we have $\text{ord}'_e(z) = \text{ord}_e(z)$. Thus for each $x \in V$, the cost paid by x to move the pointers LeftPtr/RightPtr is $|\text{ord}_e(x) - \text{ord}'_e(x)|$. ■

We will show the following result in Section 3.1.

Lemma 6 $\sum_{e \in E} \sum_{x \in V} |\text{ord}_e(x) - \text{ord}'_e(x)|$ is $O(n^{5/2})$, where ord_e is our valid topological ordering before inserting edge e and ord'_e is our valid topological ordering after inserting e .

Theorem 1, stated in Section 1, follows from Theorem 3, Lemmas 4 and 6. Also note that the space requirement of our algorithm is $O(n^2)$, since our algorithm uses only the $n \times n$ adjacency matrix M , the array A , the queues ANC, DES, and the pointers LeftPtr, RightPtr.

3.1 Proof of Lemma 6

Let $e = (u, v)$ and let ord_e be our topological ordering before inserting (u, v) and ord'_e our topological ordering after inserting (u, v) . Our algorithm for updating ord_e to ord'_e basically permutes the vertices in the subarray $A[i..j]$. The elements which get inserted into the queue ANC move to lower indexed locations in A (compared to their locations in A before e was added), elements which get inserted into the queue DES move to higher indexed locations in A , and elements which did not get inserted into either ANC or DES remain unmoved in A . So our algorithm is essentially a permutation π_e of elements that belonged to $\text{ANC} \cup \text{DES}$.

Let ANC_e denote the ordered set of all vertices that get inserted to the set ANC in Algorithms 2.1 and 2.2 (and of course, later get deleted from ANC in Algorithm 2.2) while inserting the edge e . In other words, these are the vertices w for which $\text{ord}_e(w) > \text{ord}'_e(w)$. Define DES_e as the ordered set of all those vertices w for which $\text{ord}_e(w) < \text{ord}'_e(w)$. Equivalently, these are all the vertices that get inserted into DES in Algorithms 2.1 and 2.3. Let $\text{ANC}_e = \{u_0, u_1, \dots, u_k\}$, where $u_0 = u$ and $\text{ord}(u_0) > \text{ord}(u_1) > \dots > \text{ord}(u_k)$, and let $\text{DES}_e = \{v_0, v_1, \dots, v_s\}$, where $v_0 = v$, and $\text{ord}(v_0) < \text{ord}(v_1) < \dots < \text{ord}(v_s)$.

Let us assume that all the vertices of $\text{ANC}_e \cup \text{DES}_e$ are in their old locations in A (those locations given by the ordering ord_e ; so $A[i] = v$ and $A[j] = u$). We will now decompose the permutation π_e on these elements into a composition of swaps. Note that our algorithm does not perform any swaps, however to prove Lemma 6, it is useful to view π_e as a composition of appropriate swaps. The function $\text{swap}(x, y)$ takes

as input: $x \in \text{ANC}_e$ and $y \in \text{DES}_e$, where $\text{ord}(x) > \text{ord}(y)$, and swaps the occurrences of x and y in the array A . That is, if $A[h] = x$ and $A[\ell] = y$, where $h > \ell$, then $\text{swap}(x, y)$ makes $A[\ell] = x$ and $A[h] = y$.

The intuition behind decomposing π_e into swaps between such an element $x \in \text{ANC}_e$ and such an element $y \in \text{DES}_e$ is that we will have the following useful property: *whenever we swap two elements x and y , it is always the case that $\text{ord}(x) > \text{ord}(y)$ and we will never swap x and y again in the future (while inserting other new edges) since we now have the relationship $x \rightsquigarrow y$, so $\text{ord}(y) > \text{ord}(x)$ has to hold from now on.*

We will use the symbol $\overline{\text{ord}}$ to indicate the *dynamic* inverse function of A , so that $\overline{\text{ord}}$ reflects instantly changes made in the array A . So as soon as we swap x and y so that $A[\ell] = x$ and $A[h] = y$, we will say $\overline{\text{ord}}(x) = \ell$ and $\overline{\text{ord}}(y) = h$. Thus the function $\overline{\text{ord}}$ gets initialized to the function ord_e , it gets updated with every swap that we perform and finally becomes the function ord'_e .

3.1.1 3.1.1 Decomposing π_e into appropriate swaps.

- Initialize the permutation π_e to identity and the function $\overline{\text{ord}}$ to ord_e .
- for** $x \in \{u_k, u_{k-1}, \dots, u_0\}$ (this is ANC_e : elements in reverse order of insertion into ANC_e) **do**
 - for** $y \in \{v_s, v_{s-1}, \dots, v_0\}$ (this is DES_e : elements in reverse order of insertion into DES_e) **do**
 - if** $\overline{\text{ord}}(x) > \overline{\text{ord}}(y)$ **then**
 - $\pi_e = \text{swap}(x, y) \circ \pi_e$ (*) {Note that swapping x and y changes their $\overline{\text{ord}}$ values.}
 - end if**
 - end for**
 - end for**
 - Return π_e (as a composition of appropriate swaps).

To get a better insight into this decomposition of π_e , let us take the example of the element $u_k \in \text{ANC}$ (u_k has the minimum ord_e value among all the elements in ANC). Let v_0, v_1, \dots, v_r be the elements of DES_e whose ord_e value is less than $\text{ord}_e(u_k) = \alpha$. Recall that $\text{ord}_e(v_0) < \dots < \text{ord}_e(v_r) < \text{ord}_e(v_{r+1}) < \dots < \text{ord}_e(v_s)$. When the outer **for** loop for $x = u_k$ is executed, u_k does not swap with v_s, \dots, v_{r+1} . The first element that u_k swaps with is v_r , then it swaps with v_{r-1} , so on, and u_k finally swaps with v_0 and takes the location i in A that was occupied by v_0 . Thus $\text{ord}'_e(u_k) = i$ and $\overline{\text{ord}}(v_0), \dots, \overline{\text{ord}}(v_r)$ are higher than what they were formerly, since each $v_\ell \in \{v_0, \dots, v_{r-1}\}$ is currently occupying the location that was formerly occupied by $v_{\ell+1}$, and v_r is occupying u_k 's old location α . Thus the total movement of u_k from location α to location i , can be written as:

$$\text{ord}_e(u_k) - \text{ord}'_e(u_k) = \alpha - i = \sum_{v_\ell \in \{v_r, \dots, v_0\}} d(u_k, v_\ell)$$

where $d(u_k, v_\ell) = \overline{\text{ord}}(u_k) - \overline{\text{ord}}(v_\ell)$ when $\text{swap}(u_k, v_\ell)$ is included in π_e (refer to (*) in Section 3.1.1).

Correctness of our decomposition of π_e . It is easy to see that the composition of swaps, π_e , that is returned by the above method in Section 3.1.1, when applied on $\text{ANC}_e = \{u_k, u_{k-1}, \dots, u_0\}$ and $\text{DES}_e = \{v_0, \dots, v_{s-1}, v_s\}$, results in these elements occurring in the relative order: $u_k, u_{k-1}, \dots, u_0, v_0, \dots, v_{s-1}, v_s$ in A . We claim that our algorithm (Algorithms 2.1, 2.2, 2.3) places these elements in the same order in A . This is because our algorithm maintained both ANC and DES as queues - so elements of ANC_e (similarly, DES_e) do not cross each other, so u_k, \dots, u_0 (resp., v_0, \dots, v_s) will be placed in this order, from left to right, in A . Also, we insert all elements of ANC_e in $A[i..t]$ and all elements of DES_e in $A[(t+1)..j]$, thus our algorithm puts elements of $\text{ANC}_e \cup \text{DES}_e$ in the order $u_k, u_{k-1}, \dots, u_0, v_0, \dots, v_{s-1}, v_s$ in A . Thus we have obtained a correct decomposition (into swaps) of the permutation performed by our algorithm.

For every pair $(x, y) \in \text{ANC}_e \times \text{DES}_e$, if $\text{swap}(x, y)$ is included in π_e (see (*)), define $d(x, y) = \overline{\text{ord}}(x) - \overline{\text{ord}}(y)$, the difference in the location indices occupied by x and y , when $\text{swap}(x, y)$ gets included in π_e . For instance, $d(u_k, v_0) = \text{ord}_e(v_1) - \text{ord}_e(v_0)$ since u_k is in the location $\text{ord}_e(v_1)$ (due to swaps with v_r, \dots, v_1) and v_0 is unmoved in its original location $\text{ord}_e(v_0)$, at the instant when $\text{swap}(u_k, v_0)$ gets included in π_e .

Since we broke the total movement in A of any $x \in \text{ANC}_e$ (which is $\text{ord}_e(x) - \text{ord}'_e(x)$) into a sequence of swaps with certain elements in DES_e , we have for any $x \in \text{ANC}_e$

$$\text{ord}_e(x) - \text{ord}'_e(x) = \sum_{y:(x,y) \in \pi_e} d(x,y),$$

where we are using “ $(x,y) \in \pi_e$ ” to stand for “ $\text{swap}(x,y)$ exists in π_e ”.

$$\text{We have } \sum_{x \in V} |\text{ord}_e(x) - \text{ord}'_e(x)| = \sum_{w \in \text{ANC}_e} (\text{ord}_e(w) - \text{ord}'_e(w)) + \sum_{y \in \text{DES}_e} (\text{ord}'_e(y) - \text{ord}_e(y)) \quad (1)$$

$$= 2 \sum_{x \in \text{ANC}_e} (\text{ord}_e(x) - \text{ord}'_e(x)) \quad (2)$$

$$= 2 \sum_{x \in \text{ANC}_e} \sum_{y:(x,y) \in \pi_e} d(x,y) \quad (3)$$

$$= 2 \sum_{(x,y):(x,y) \in \pi_e} d(x,y). \quad (4)$$

Equality (2) follows from (1) because $\sum_{x \in V} \text{ord}(x) = \sum_{x \in V} \text{ord}'(x)$. Equality (3) follows from the preceding paragraph. So the entire running time to insert all edges in E is $2 \sum_{e \in E} \sum_{(x,y):(x,y) \in \pi_e} d(x,y)$.

We now claim that for any pair (x,y) , we can have $(x,y) \in \pi_e$ for at most one permutation π_e . For $\text{swap}(x,y)$ to exist in π_e , we need (i) $(x,y) \in \text{ANC}_e \times \text{DES}_e$, and (ii) $\text{ord}(x) > \text{ord}(y)$. Once π_e swaps x and y , subsequently $x \rightsquigarrow y$ (since $x \rightsquigarrow u \rightarrow v \rightsquigarrow y$) and $\text{ord}(y) > \text{ord}(x)$, so (x,y) can never again satisfy $\text{ord}(x) > \text{ord}(y)$. So for any pair (x,y) , $\text{swap}(x,y)$ can occur in at most one permutation π_e over all $e \in E$. Thus we have:

$$\sum_{e \in E} \sum_{(x,y):(x,y) \in \pi_e} d(x,y) = \sum_{(x,y):(x,y) \in \pi_e \text{ for some } e} d(x,y). \quad (5)$$

Note that the summation on the right hand side in Inequality (5) is over all those pairs $(x,y) \in V \times V$ such that $\text{swap}(x,y)$ exists in some π_e , for $e \in E$.

The following lemma was shown in [1]³. This finishes the proof of Lemma 6.

Lemma 7 $\sum d(x,y)$ is $O(n^{5/2})$, where the summation is over all those pairs (x,y) such that $\text{swap}(x,y)$ exists in some permutation $\pi_e, e \in E$.

Proof: We present the proof of this lemma given in [1]. We need to show that $\sum_{x,y} d(x,y)$ is $O(n^{5/2})$. Let ord^* denote the final topological ordering. Define

$$X(\text{ord}^*(x), \text{ord}^*(y)) = \begin{cases} d(x,y) & \text{if there is some permutation } \pi_e \text{ that swaps } x \text{ and } y \\ 0 & \text{otherwise.} \end{cases}$$

Since $\text{swap}(x,y)$ can occur in at most one permutation π_e , the variable $X(i,j)$ is clearly defined. Next, we model a few linear constraints on $X(i,j)$, formulate it as a linear program and use this LP to prove that $\max\{\sum_{i,j} X(i,j)\} = O(n^{5/2})$. By definition of $d(x,y)$ and $X(i,j)$,

$$0 \leq X(i,j) \leq n, \text{ for all } i, j \in \{1 \dots n\}.$$

For $j \leq i$, the corresponding edges $(\text{ord}^{*-1}(i), \text{ord}^{*-1}(j))$ go backwards and thus are never inserted at all. Consequently,

$$X(i,j) = 0 \text{ for all } j \leq i.$$

Now consider an arbitrary vertex w , which is finally at position i , i.e., $\text{ord}^*(w) = i$. Over the insertion of all the edges, this vertex has been moved left and right via swapping with several other vertices. Strictly

³The algorithm in [1] performs swaps to obtain a valid topological ordering and Lemma 7 is used in their analysis to show an $O(n^{2.75})$ upper bound for the running time of their algorithm.

speaking, it has been swapped left with vertices at final positions $j > i$ and has been swapped right with vertices at final position $j < i$. Hence, the overall movement to the left is $\sum_{j>i} X(i, j)$ and to the right is $\sum_{j<i} X(j, i)$. Since the net movement (difference between the final and the initial position) must be less than n ,

$$\sum_{j>i} X(i, j) - \sum_{j<i} X(j, i) \leq n \text{ for all } 1 \leq i \leq n.$$

Putting all the constraints together, we aim to solve the following linear program.

$$\max \sum_{1 \leq i \leq n, 1 \leq j \leq n} X(i, j) \text{ such that}$$

- (i) $X(i, j) = 0$ for all $1 \leq i \leq n$ and $1 \leq j \leq i$
- (ii) $0 \leq X(i, j) \leq n$ for all $1 \leq i \leq n$ and $i < j \leq n$
- (iii) $\sum_{j>i} X(i, j) - \sum_{j<i} X(j, i) \leq n - 1$, for all $1 \leq i \leq n$

In order to prove the upper bound on the solution to this LP, we consider the dual problem:

$$\min \left[n \sum_{0 \leq i \leq n, i < j < n} Y_{i-n+j} + n \sum_{0 \leq i < n} Y_{n^2+i} \right] \text{ such that}$$

- (i) $Y_{i-n+j} \geq 1$ for all $0 \leq i < n$ and for all $j \leq i$
- (ii) $Y_{i-n+j} + Y_{n^2+i} - Y_{n^2+j} \geq 1$ for all $0 \leq i < n$ and for all $j > i$
- (iii) $Y_i \geq 0$ for all $0 \leq i < n^2 + n$

and the following feasible solution for the dual:

$$\begin{aligned} Y_{i-n+j} &= 1 \text{ for all } 0 \leq i < n \text{ and for all } 0 \leq j \leq i \\ Y_{i-n+j} &= 1 \text{ for all } 0 \leq i < n \text{ and for all } i < j \leq i + 1 + 2\sqrt{n} \\ Y_{i-n+j} &= 0 \text{ for all } 0 \leq i < n \text{ and for all } j > i + 1 + 2\sqrt{n} \\ Y_{n^2+i} &= \sqrt{n-i} \text{ for all } 0 \leq i < n. \end{aligned}$$

The solution has a value of $n^2 + 2n^{5/2} + n \sum_{i=1}^n \sqrt{i} = O(n^{5/2})$, which by the primal-dual theorem is a bound on the solution of the original LP. This completes the proof of Lemma 7 and thus Lemma 6 is proved. \blacksquare

4 The $O((m + n \log n)\sqrt{m})$ algorithm

In this section we present an algorithm with running time $O((m + n \log n)\sqrt{m})$ for online topological ordering. This algorithm is an adaptation of the algorithm by Katriel and Bodlaender in [6] and uses the *Ordered List* data structure from [5], also used in [6] for this problem. That is, the function ord on V is maintained by a data structure ORD which is a data structure that allows a total order to be maintained over a list of items. Each item x in ORD has an associated integer label $\text{ord}(x)$ and the label associated with x is smaller than the label associated with y , iff x precedes y in the total order. The following operations can be performed in constant amortized time [see Dietz and Sleator [5], Bender et al. [3] for details]: the query $\text{Order}(x, y)$ determines whether x precedes y or y precedes x in the total order (i.e., if $\text{ord}(x) < \text{ord}(y)$ or $\text{ord}(y) < \text{ord}(x)$), $\text{InsertAfter}(x, y)$ ($\text{InsertBefore}(x, y)$) inserts the item x immediately after (before) the item y in the total order, and $\text{Delete}(x)$ removes the item x .

When a new edge (u, v) is added to a graph G , there are two cases: (i) either $\text{Order}(u, v)$ is true, in which case the current ordering of elements in ORD is still a valid ordering, so we need to do nothing except add

(u, v) in the list of edges incoming into u and in the list of edges going out of v ; (ii) $Order(u, v)$ is false, in which case the edge (u, v) is invalidating and we need to change the order of vertices in ORD .

Our algorithm to insert an invalidating edge (u, v) performs various steps. Each step involves visiting an ancestor of u and/or visiting a descendant of v .

- Initially u is the only ancestor of u that we know. So we *visit* u . We use a Fibonacci heap F_u to store ancestors of u that we have seen but not yet visited. For an ancestor x of u , $visit(x)$ means that for every edge (w, x) into x , we check if w is already present in F_u and if w is not present in F_u , we insert w into F_u .

- The next ancestor of u that we visit is the vertex with the maximum ORD label in F_u . An *extract-max* operation on this F-heap (the priority of vertices in F_u is determined by how *high* their associated label is in ORD) determines this vertex x .

- Analogously, we have a Fibonacci heap F_v to store descendants of v that we have seen but not yet visited. For any descendant y of v , $visit(y)$ means that for every edge (y, z) out of y , we check if z is already present in the F-heap F_v and if z is not present in F_v , we insert z into F_v . The priority of vertices in F_v is determined by how *low* their associated label is in ORD . Thus an *extract-min* operation on this F-heap determines the next descendant of v that we visit.

- At the end of each step we check if $Order(x, y)$ is true, where x is the last extracted vertex from F_u and y is the last extracted vertex from F_v . If $Order(x, y)$ is true (i.e., if x precedes y in ORD), then this is the termination step; all the ancestors of u that we visited, call them $\{u_0(=u), \dots, u_k\}$ and the descendants of v that we visited, call them $\{v_0(=v), \dots, v_s\}$, get reinserted in ORD after x or before y , in the order $u_k, \dots, u_0, v_0, \dots, v_s$. Else, i.e., if y precedes x in ORD , then we delete x and y from their current positions in ORD and in the next step we either visit x or y or both x and y .

In any step of the algorithm, if $\{u_0, u_1, \dots, u_r\}$ is the set of ancestors of u that we have already visited (in this order, so $ord(u_r) < \dots < ord(u_0)$) in the previous steps, then the ancestor of u that we plan to visit next is the vertex x with the *maximum* ORD label that has an edge into a vertex in $\{u_0, u_1, \dots, u_r\}$. Once we visit x , we would have visited all ancestors of u with ORD labels sandwiched between $ord(x)$ and $ord(u)$. Similarly, on the side of v , if v_0, v_1, \dots, v_ℓ are the descendants of v that we have already visited (i.e., $ord(v_0) < \dots < ord(v_\ell)$), then the descendant of v that we plan to visit next is the vertex y with the *minimum* ORD label which has an edge coming from one of $\{v_0, v_1, \dots, v_\ell\}$.

When $Order(x, y)$ is true, it means that we have discovered *all* descendants of v with ORD label values between $ord(v)$ and i , and *all* ancestors of u with ORD label values between i and $ord(u)$ (where i is any value such that $ord(x) \leq i \leq ord(y)$). Thus we can relocate vertices $u_k, \dots, u_0, v_0, \dots, v_s$ (in this order) between x and y . It is easy to see that now for every $(a, b) \in E$, we have that a precedes b in ORD .

What remains to be explained is how to make the choice between the following 3 options in each step: (i) visit(x) and visit(y), (ii) only visit(x), or (iii) only visit(y).

Visit(x) and/or Visit(y). In order to make the choice between visit(x) and/or visit(y), let us make the following definitions: Let ANC denote the set of ancestors of u that we have already visited plus the ancestor x that we plan to visit next. Let DES denote the set of descendants of v that we have already visited plus the descendant y that we plan to visit next. Let m_D be the sum of out-degrees of vertices in DES and let m_A be the sum of in-degrees of vertices in ANC.

If we were to visit x in the current step, then the *total* work done by us on the side of u so far would be $m_A + |ANC| \log n$ (to have examined m_A edges incoming into ANC and for at most m_A insertions in F_u , and to have performed $|ANC|$ many *extract-max* operations on F_u). Similarly, if we were to visit y in the current step, then the total work done by us on the side of v so far would be $m_D + |DES| \log n$.

Definition 1 If $m_A \leq m_D \leq m_A + |ANC| \log n$ or $m_D \leq m_A \leq m_D + |DES| \log n$ then we say that m_A and m_D are “balanced” with respect to each other. Else we say that they are not balanced with respect to each other.

If m_A and m_D are balanced with respect to each other, then we visit both x and y . Else if $m_A < m_D$, then we visit only x , otherwise we visit only y . This is the difference between our algorithm and the algorithm

in [6] - in the latter algorithm, either only x is visited or only y is visited unless $m_D = m_A$, in which case both x and y are visited. In our algorithm we are ready to visit both x and y more often, that is, whenever m_A and m_D are “more or less” equal to each other. If we visit both x and y , then the total work done is $m_A + |\text{ANC}| \log n + m_D + |\text{DES}| \log n$. We can give a good upper bound for this quantity using the fact that m_A and m_D are balanced with respect to each other. On the other hand, if m_A and m_D are not balanced w.r.t. each other, we are not able to give a good upper bound for the *sum* of $(m_A + |\text{ANC}| \log n)$ and $(m_D + |\text{DES}| \log n)$, hence we visit either x or y , depending upon the smaller value in $\{m_A, m_D\}$.

4.1 The algorithm

Our entire algorithm to reorder vertices in *ORD* upon the insertion of an invalidating edge (u, v) is described as Algorithm 4.1. This algorithm is basically an implementation of what was described in the previous section with a check at the beginning of every step to see if m_A and m_D are balanced with respect to each other or not. If they are, then we visit both x and y . Else, we visit only one of them (x if $m_A < m_D$, else y). The algorithm maintains the invariant that the *ORD* labels of all elements in *ANC* are higher than the *ORD* labels of all elements in *DES*. The termination condition is determined by $\text{Order}(x, y)$ being true, where x is the last extracted vertex from F_u and y is the last extracted vertex from F_v .

For simplicity, in the description of the algorithm we assumed that the heaps F_u and F_v remain non-empty (otherwise extract-max/extract-min operations would return null values) - handling these cases is easy. We also assumed that the edges inserted are the edges of a DAG. Hence we did not perform any cycle detection here. (Cycle detection can be easily incorporated, by using 2 flags for each vertex that indicate its membership in F_u and in F_v .) When our algorithm terminates, it is easy to see the order of vertices in *ORD* is a valid topological ordering. We discuss the running time of Algorithm 4.1 in the next section.

4.1.1 The running time

Let $T(e)$ denote the time taken by Algorithm 4.1 to insert an edge e . We need to show an upper bound for $\sum_e T(e)$, where the sum is over all invalidating edges e . For simplicity of exposition, let us define the following modes. While inserting an edge (u, v) , if a step of our algorithm involved visiting an ancestor of u and a descendant of v , we say that step was performed in *mode (i)*. That is, at the beginning of that step, we had m_A and m_D balanced with respect to each other. If a step involved visiting only an ancestor of u , then we say that the step was performed in *mode (ii)*, else we say that the step was performed in *mode (iii)*.

We partition the sum $\sum_e T(e)$ into 2 parts depending upon the *mode* of the termination step of our algorithm. Let $S_1 = \sum_e T(e)$ be the time taken by our algorithm over all those edges e such that the termination step was performed in mode (i). Let $S_2 = \sum_e T(e)$ where the sum is over all those edges e such that the termination step was performed in mode (ii) or mode (iii). We will show that both S_1 and S_2 are $O((m + n \log n) \sqrt{m})$. These bounds on S_1 and S_2 will prove Theorem 2 stated in Section 1.

The following lemma shows the bound on S_1 . We then show an analogous bound on S_2 .

Lemma 8 S_1 is $O((m + n \log n) \sqrt{m})$.

Proof: Let us consider any particular edge $e_i = (u, v)$ such that the last step of Algorithm 4.1 while inserting e_i was performed in mode (i). So the termination step involved visiting an ancestor u_k of u , extracting the next ancestor x of u , visiting a descendant v_s of v , extracting the next descendant y of v and then checking that x precedes y in *ORD*.

Let the set $\text{ANC} = \{u, u_1, \dots, u_k\}$ and the set $\text{DES} = \{v, v_1, \dots, v_s\}$. Let m_A be the sum of in-degrees of vertices in *ANC* and let m_D be the sum of out-degrees of vertices in *DES*. During all the steps of the algorithm, we extracted $|\text{ANC}|$ many vertices (the vertices u_1, \dots, u_k and x) from F_u and $|\text{DES}|$ many vertices from F_v . So we have $T(e_i)$ is $O(m_A + |\text{ANC}| \log n + m_D + |\text{DES}| \log n)$.

Since the termination step was performed in mode (i), we have that $m_A \leq m_D \leq m_A + |\text{ANC}| \log n$ or $m_D \leq m_A \leq m_D + |\text{DES}| \log n$. Without loss of generality let us assume that $m_A \leq m_D \leq m_A + |\text{ANC}| \log n$.

Algorithm 4.1 Our algorithm to reorder vertices in *ORD* upon insertion of an invalidating edge (u, v) .

Initially, $ANC = \{u\}$ and $DES = \{v\}$.

Set $x = u$ and $y = v$. Delete x and y from their current locations in *ORD*.

Set $m_A = u$'s in-degree and $m_D = v$'s out-degree.

while TRUE **do**

if m_A and m_D are balanced w.r.t. each other (see Defn. 1) **then**

 Visit(x) and Visit(y).

 {So new vertices get inserted into F_u and into F_v .}

$x = \text{extract-max}(F_u)$

$y = \text{extract-min}(F_v)$

if $\text{ord}(x) < \text{ord}(y)$ **then**

 – insert all elements of *ANC* (with the same relative order within themselves) followed by all elements of *DES* (with the same relative order) after x in *ORD*

break {This terminates the while loop}

else

 Delete x and y from their current positions in *ORD*.

$ANC = ANC \cup \{x\}$ and $DES = DES \cup \{y\}$

$m_A = m_A + x$'s in-degree and $m_D = m_D + y$'s out-degree

end if

else if $m_A < m_D$ **then**

 Visit(x)

$x = \text{extract-max}(F_u)$

if $\text{ord}(x) < \text{ord}(y)$ **then**

 – insert all elements of *ANC* followed by all elements of *DES* after x in *ORD*.

break

else

 Delete x from its current position in *ORD*.

 Set $ANC = ANC \cup \{x\}$ and $m_A = m_A + x$'s in-degree.

end if

else

 Visit(y)

 Let $y = \text{extract-max}(F_v)$

if $\text{ord}(x) < \text{ord}(y)$ **then**

 – insert all elements of *ANC* followed by all elements of *DES* before y in *ORD*.

break

else

 Delete y from its current position in *ORD*.

 Set $DES = DES \cup \{y\}$ and $m_D = m_D + y$'s out-degree.

end if

end if

end while

Hence $T(e_i)$ can be upper bounded by some constant times

$$m_A + |ANC| \log n + |DES| \log n. \quad (6)$$

Let us assume that $|ANC| > |DES|$. (Note that the case $|ANC| < |DES|$ is symmetric to this and the case $|ANC| = |DES|$ is the easiest.) Since the termination step was performed in mode (i), for $|ANC|$ to be larger than $|DES|$, it must be the case that at some point in the past, our algorithm to insert e_i was operating in mode (ii) and that contributed to accumulating quite a few ancestors of u . Let step t be the last step

that was operated in mode (ii). So at the beginning of step t we had $m'_A + |\text{ANC}'| \log n \leq m'_D$, where ANC' was the set of ancestors of u extracted from the F-heap F_u till the beginning of step t and m'_A is the sum of in-degrees of vertices in ANC' , and m'_D is the sum of out-degrees of vertices in DES' where DES' was the set of descendants of v extracted from the F-heap F_v till the beginning of step t . After step t , we never operated our algorithm in mode (ii). Thus subsequent to step t whenever we extracted a vertex from F_u , we also extracted a corresponding vertex from F_v . So we have $|\text{ANC}| \leq |\text{ANC}'| + |\text{DES}|$. Using this inequality in (6), we get that

$$T(e_i) \leq c(m_A + |\text{DES}| \log n + |\text{ANC}'| \log n), \quad \text{for some constant } c.$$

Claim 1 *We have the following relations:*

- $m_A^2 \leq m_A \cdot m_D \leq \Phi(e_i)$
where $\Phi(e_i)$ is the number of pairs of edges (e, e') for which the relationship $e \rightsquigarrow e'$ has started now for the first time due to the insertion of e_i . [We say $(a, b) \rightsquigarrow (c, d)$ if b is an ancestor of c .]
- $|\text{DES}|^2 \leq |\text{ANC}| \cdot |\text{DES}| \leq N(e_i)$
where $N(e_i)$ is the number of pairs of vertices (w, w') such that $w \rightsquigarrow w'$ has started now for the first time due to the insertion of e_i . [We say $w \rightsquigarrow w'$ if w is an ancestor of w' .]
- $(|\text{ANC}'| \log n)^2 \leq (|\text{ANC}'| \log n) \cdot m'_D \leq \Psi(e_i) \log n$
where $\Psi(e_i)$ is the number of pairs $(w, e) \in V \times E$ for which the relationship $w \rightsquigarrow e$ has started now for the first time. [We say $w \rightsquigarrow (a, b)$ if w is an ancestor of a .]

Proof of Claim 1. After the insertion of edge (u, v) we have $e \rightsquigarrow e'$ for every edge e incoming into $\text{ANC} = \{u, u_1, \dots, u_k\}$ and every edge e' outgoing from $\text{DES} = \{v, v_1, \dots, v_s\}$. Prior to inserting e_i , the *sink* of each of the m_A edges incoming into ANC has a higher *ORD* label compared to the *source* of each of the m_D edges outgoing from DES - thus we could have had no relation of the form $e \rightsquigarrow e'$ between the m_A edges incoming into ANC and the m_D edges outgoing from DES . So $\Phi(e_i) \geq m_A m_D$.

The above argument also shows that $N(e_i) \geq |\text{ANC}| \cdot |\text{DES}|$. We have $|\text{ANC}'| m'_D \leq \Psi(e_i)$ because the source of each of these m'_D edges had a lower *ORD* label than the vertices in ANC' prior to inserting e_i ; thus the relation $w \rightsquigarrow e'$ for each $w \in \text{ANC}'$ and the edges e' (m'_D many of them) outgoing from DES' is being formed for the first time now. ■

Now we are ready to finish the proof of Lemma 8. Corresponding to the insertion of each edge e_j whose termination step was in mode (i), the work done by our algorithm is at most $c(f_j + g_j + h_j)$ where $f_j^2 \leq \Phi(e_j)$ and $g_j^2 \leq N(e_j) \log^2 n$ and $h_j^2 \leq \Psi(e_j) \log n$. In order to bound $\sum_j (f_j + g_j + h_j)$, we use Cauchy's inequality which states that $\sum_{i=1}^m x_i \leq \sqrt{\sum_i x_i^2} \sqrt{m}$, for $x_1, \dots, x_m \in \mathbb{R}$. This yields

$$\sum_j f_j + g_j + h_j \leq (\sqrt{\sum_j f_j^2} + \sqrt{\sum_j g_j^2} + \sqrt{\sum_j h_j^2}) \sqrt{m} \quad (7)$$

$$\leq \left(\sqrt{\sum_j \Phi(e_j)} + \sqrt{\sum_j N(e_j) \log n} + \sqrt{\sum_j \Psi(e_j) \log n} \right) \sqrt{m} \quad (8)$$

$$\leq (m + n \log n + \sqrt{mn \log n}) \sqrt{m}. \quad (9)$$

We have $\sum_j \Phi(e_j)$ is at most $\binom{m}{2}$ since each pair of edges e and e' can contribute at most 1 to $\sum_j \Phi(e_j)$; similarly $\sum_j \Psi(e_j)$ is at most mn , and $\sum_j N(e_j)$ is at most $\binom{n}{2}$. This yields Inequality (9) from (8). Since $\sqrt{mn \log n} \leq (m + n \log n)/2$, this completes the proof that the sum S_1 is $O((m + n \log n) \sqrt{m})$. ■

Analogous to Lemma 8, we need to show the following lemma in order to bound the running time of Algorithm 4.1 by $O((m + n \log n) \sqrt{m})$.

Lemma 9 S_2 is $O((m + n \log n) \sqrt{m})$.

Proof: Recall that $S_2 = \sum T(e)$ where the sum is over all those e such that the termination step of Algorithm 4.1 was performed in mode (ii) or mode (iii). Let us further partition this sum into $\sum_e T(e)$ over all those e for which the last step was performed in mode (ii) and $\sum_{e'} T(e')$ over all those e' for which the last step was performed in mode (iii). The analysis for the second sum will be entirely symmetric to the first. We will now bound the first sum.

Let $e_i = (u, v)$ be an edge such that the termination step of our algorithm was performed in mode (ii). Let the set $\text{ANC} = \{u, u_1, \dots, u_k\}$ and let the set $\text{DES} = \{v, v_1, \dots, v_s\}$. Let m_A be the sum of in-degrees of vertices in ANC and let m_D be the sum of out-degrees of vertices in DES . Since the termination step was performed in mode (ii), we have $m_A + |\text{ANC}| \log n < m_D$.

The work that we did in all the steps while inserting (u, v) from the side of the vertex u is $m_A + |\text{ANC}| \log n$. Let step t be the last step of our algorithm which was operated in mode (i) or in mode (iii). If there was no such step, then the total work done is at most $m_A + |\text{ANC}| \log n$ and it is easy to bound this using the inequality $m_A + |\text{ANC}| \log n < m_D$. Hence, let us assume that such a step t did exist and let DES' be the set of descendants of v at the beginning of step t and let m'_D be the sum of out-degrees of vertices in DES' . Note that we have $m'_D \leq m'_A + |\text{ANC}'| \log n$ since this step was operated in mode (i) or in mode (iii).

The total work done from the side of v is $m'_D + |\text{DES}'| \log n$. Thus the total work $T(e_i)$ is $O(m_A + |\text{ANC}| \log n + m'_v + |\text{DES}'| \log n)$. Using the inequality $m'_D \leq m'_A + |\text{ANC}'| \log n \leq m_A + |\text{ANC}| \log n$, we have $T(e_i)$ upper bounded by a constant times

$$m_A + |\text{ANC}| \log n + |\text{DES}'| \log n. \quad (10)$$

Let us concentrate on the last term $|\text{DES}'| \log n$ in the above sum. Let DES'' be the set of descendants of v at the beginning of the last step when we ran in mode (iii). So $m''_D + |\text{DES}''| \log n \leq m''_A$, where m''_D is the sum of the out-degrees of vertices in DES'' , ANC'' is the set of ancestors of u at the beginning of this step and m''_A is the sum of the in-degrees of vertices in ANC'' . After this step, whenever we explored edges on the side of v , it was in mode (i), thus visiting a descendant of v was always accompanied by visiting an ancestor of u . So $|\text{DES}'| \leq |\text{DES}''| + |\text{ANC}|$. Substituting this bound in (10) we get that

$$T(e_i) \leq c(m_A + |\text{ANC}| \log n + |\text{DES}''| \log n) \quad \text{for some constant } c.$$

We have the following relations (see Claim 1 for the definitions of $\Phi(e_i)$ and $\Psi(e_i)$):

- $(m_A + |\text{ANC}| \log n)^2 \leq (m_A + |\text{ANC}| \log n) m_D$
 $\leq \Phi(e_i) + \Psi(e_i) \log n.$
- $(|\text{DES}''| \log n)^2 \leq (|\text{DES}''| \log n) m''_A$
 $\leq v(e_i) \log n$

where $v(e_i)$ is the number of pairs $(e, z) \in E \times V$ that get ordered with respect to each other for the first time now due to the insertion of e_i .

The proofs of the above relations are analogous to the proofs given in Claim 1 and we refer the reader to the proof of Claim 1 (in Section 4.1.1).

We are now ready to complete the proof of Lemma 9. $\sum T(e_i)$ where the sum is over all those e_i whose last step was performed in mode (ii) is at most $\sum_i (p_i + q_i)$ where $p_i^2 \leq \Phi(e_i) + \Psi(e_i) \log n$ and $q_i^2 \leq v(e_i) \log n$. Note that $\sum_i v(e_i)$ is at most mn since each pair $(e, z) \in E \times V$ can contribute at most 1 to $\sum_i v(e_i)$. Using Cauchy's inequality, we have

$$\begin{aligned} \sum_i (p_i + q_i) &\leq (\sqrt{\sum p_i^2} + \sqrt{\sum q_i^2}) \sqrt{m} \\ &\leq \left(\sqrt{\sum \Phi(e_i) + \sum \Psi(e_i) \log n} + \sqrt{\sum v(e_i) \log n} \right) \sqrt{m} \\ &\leq (\sqrt{m^2 + mn \log n} + \sqrt{mn \log n}) \sqrt{m} \\ &\leq (m + 2\sqrt{mn \log n}) \sqrt{m} \end{aligned}$$

Analogously, we can show that $\sum T(e_i)$ where the sum is over all those e_i whose last step was performed in mode (iii) is at most $O((m + \sqrt{mn \log n})\sqrt{m})$. Thus S_2 is $O((m + \sqrt{mn \log n})\sqrt{m})$. Since $\sqrt{mn \log n} \leq (m + n \log n)/2$ (geometric mean is at most the arithmetic mean), we have S_2 is $O((m + n \log n)\sqrt{m})$. ■

Conclusions. We considered the problem of maintaining the topological order of a directed acyclic graph on n vertices under an online edge insertion sequence of m edges. This problem has been well-studied and the previous best upper bound for this problem was $O(\min\{m^{3/2} \log n, m^{3/2} + n^2 \log n, n^{2.75}\})$. Here we showed an improved upper bound of $O(\min(n^{5/2}, (m + n \log n)\sqrt{m}))$ for this problem.

Acknowledgments. We are grateful to Deepak Ajwani and Tobias Friedrich for their helpful feedback.

References

- [1] D. Ajwani, T. Friedrich, and U. Meyer. An $O(n^{2.75})$ algorithm for online topological ordering. In *Proceedings of the 10th Scandinavian Workshop on Algorithm Theory*, LNCS 4059: 53-64, 2006. (Longer version in <http://arxiv.org/abs/cs/0602073>).
- [2] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the first annual ACM-SIAM Symposium on Discrete Algorithms*: 32-42, 1990.
- [3] Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*: 152-164, 2002.
- [4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1989.
- [5] P. Dietz and D. Sleator Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Conference on Theory of Computing*: 365-372, 1987.
- [6] I. Katriel and H. L. Bodlaender. Online topological ordering. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*: 443-450, 2005.
- [7] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59(1):53:58, 1996.
- [8] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. On-line graph algorithms for incremental compilation. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 790: 70-86, 1993.
- [9] Stephen M. Omohundro, Chu-Cheow Lim, and Jeff Bilmes. The sather language compiler/debugger implementation. *Technical Report TR-92-017, International Computer Science Institute, Berkeley*, 1992.
- [10] D.J.Pearce and P.H.J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proceedings of the Workshop on Efficient and Experimental Algorithms*, LNCS 3059: 383-398, 2004.
- [11] D.J.Pearce, P.H.J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the third international IEEE Workshop on Source Code Analysis and Manipulation*, 2003.
- [12] G. Ramalingam and T. W. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Information Processing Letters*, 51:155-161, 1994.